본 수업의 주 교재는 Silberschatz, Galvin, Gagne, "Operating System Concepts Essentials 2nd ed.", Wiley, 또는 한글번역본인 박민규, 조유근, "Operating System Concepts 에센셜 2판", 홍릉과학출판사 입니다. 본 강의 동영상의 슬라이드는 이 책의 홈페이지에서 제공하는 것을 사용했음을 밝힙니다 (https://codex.cs.yale.edu/avi/os-book/OSE2/slide-dir/index.html). 다만 강의의 편의를 위해 내용 변경 없이 슬라이드 레이아웃을 변경하였고, 진도 관리에 필요한 경우 일부 슬라이드는 생략하였습니다.

Chapter 6: CPU Scheduling

Introduction to CPU scheduling

Various CPU scheduling algorithm

Evaluation criteria for selecting a CPU-scheduling algorithm

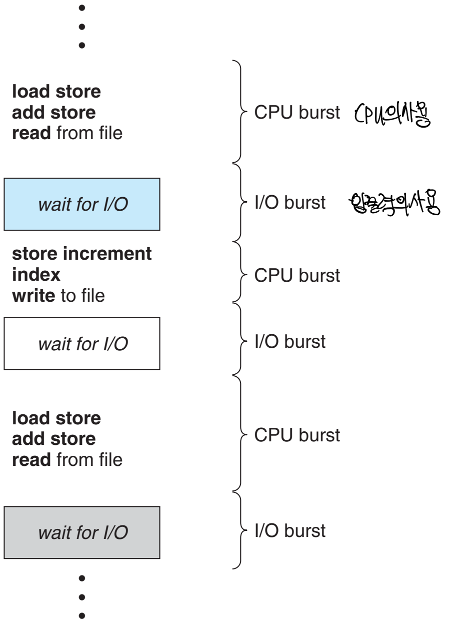
scheduling algorithms of several operating systems

Contents

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

6.1 Basic Concepts

- CPU burst–I/O Burst Cycle : Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



load store
add store
read from file

CPU burst CPU의사용

wait for I/O

I/O burst 입출력의사용

store increment
index
write to file

CPU burst

wait for I/O

I/O burst

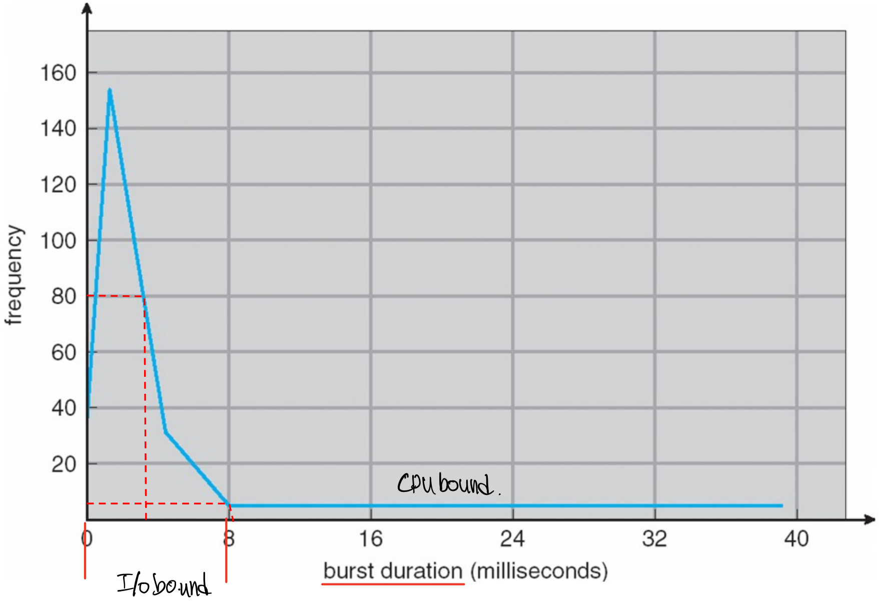
load store
add store
read from file

CPU burst

wait for I/O

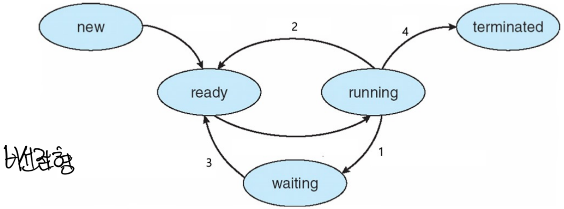
I/O burst

Histogram of CPU-burst Times



CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them ready queue 줄세우기.
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is nonpreemptive 비선점형
- All other scheduling is preemptive 선점형.
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

6.2 Scheduling Criteria

cpu scheduling의 성능지표

1. **CPU utilization** – keep the CPU as busy as possible
2. **Throughput** – # of processes that complete their execution per time unit
3. **Turnaround time** – amount of time to execute a particular process
4. **Waiting time** – amount of time a process has been waiting in the ready queue
5. **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

목표.

- Max CPU utilization ↑
- Max throughput ↑
- Min turnaround time ↓
- Min waiting time ↓
- Min response time ↓

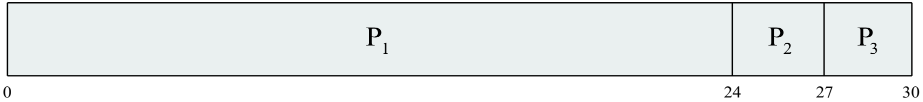
6.3 Scheduling Algorithms

- First- Come, First-Served (FCFS) Scheduling 비선점형,

  온순서대로 처리

Process	Burst Time
P_1	24
P_2	3
P_3	3

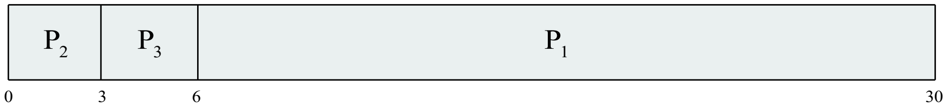
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

- Suppose that the processes arrive in the order: P_2, P_3, P_1
- The Gantt chart for the schedule is:



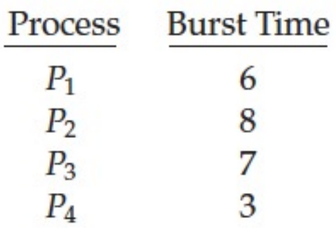
- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Much better than previous case
- Convoy effect - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user (대안)

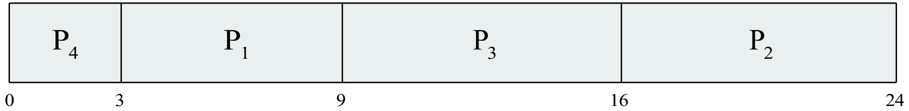
→ 이나 어렵다.

Example of SJF



Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

이 SJF 가 가장 짧다.

Determining Length of Next CPU Burst

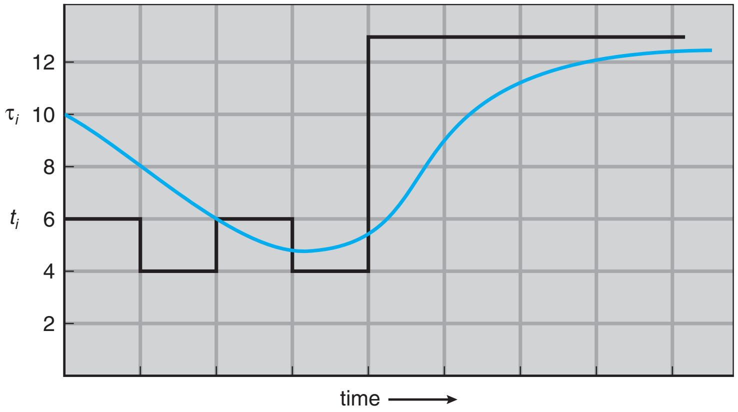
예측이 좋을수록 성능 좋아.

- Can only estimate the length – should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging 지수평균을사용

1. t_n = actual length of n^{th} CPU burst 이전에 예측된 값.
2. $\underline{\tau_{n+1}}$ = predicted value for the next CPU burst 예측할 값.
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n+1} = \underline{\alpha} t_n + \underline{(1-\alpha)}\tau_n$

- Commonly, α set to ½

Prediction of the Length of the Next CPU Burst



예측값이 실제값과
얼추 비슷해진다.

CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Examples of Exponential Averaging

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$ 상수화.
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha\ t_n$
  - Only the actual last CPU burst counts
- If we expand the formula, we get:

$$\tau_{n+1} = \alpha\ t_n + (1 - \alpha)\alpha\ t_{n-1} + \dots$$
$$+ (1 - \alpha)^j \alpha\ t_{n-j} + \dots$$
$$+ (1 - \alpha)^{n+1} \tau_0$$

뒤로갈수록 측정값에 영향이 줄어든다

- Since both α and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

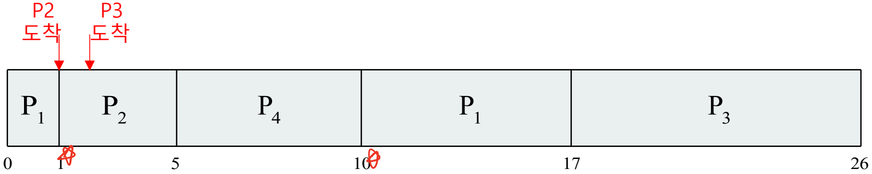
Example of Shortest-remaining-time-first

SJF의 선점형.

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	Arrival Time	Burst Time
P_1	0	8 7
P_2	1	4 4
P_3	2	9
P_4	3	5

- Preemptive SJF Gantt Chart



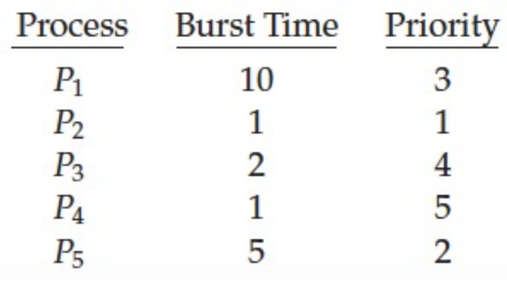
- Average waiting time = $[\underbrace{(0-0)+(10-1)}_{P_1}+(1-1)+\underbrace{(17-2)}_{P_4}+(5-3)]/4$ = 26/4 = 6.5 msec

Priority Scheduling

- A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
- Preemptive vs. nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute

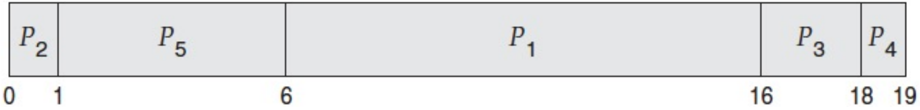
  낮은우선순위가 계속 실행X
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling



Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



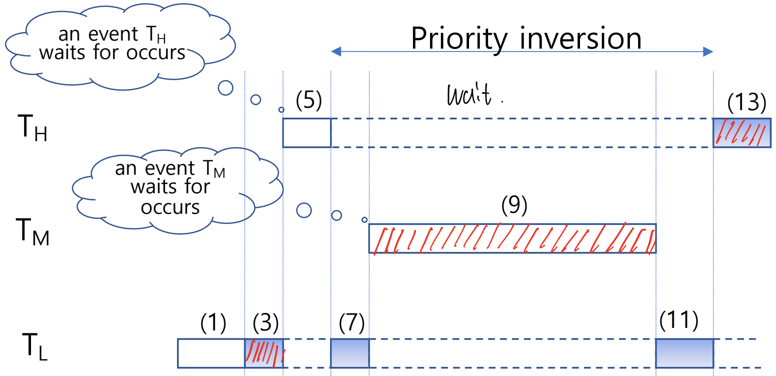
- Average waiting time = 8.2 msec

0+1+6+16+18 /5

Priority Inversion

CPU 사용과 자원 사용에서 우선순위가 역전되는 현상.

역전현상


an event T_H waits for occurs
Priority inversion
wait.
(5)
(13)
T_H
an event T_M waits for occurs
(9)
T_M
(1)
(3)
(7)
(11)
T_L

(2) T_L gets semaphore

(4) T_H preempts T_L

(6) T_H tries to get semaphore

T_L 이 가진것을 뺏으려고함.

run → wait

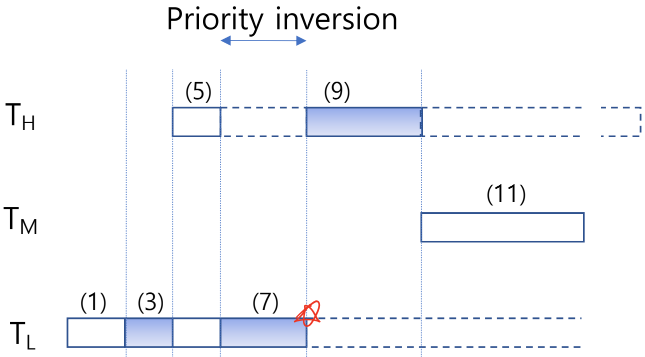
(8) T_M preempts T_L

(10) T_L resumes

(12) T_L releases the semaphore

Priority Inheritance (Donation) Protocol

해결책 상속 .



(2) T$_L$ gets semaphore

(4) T$_H$ preempts T$_L$

(6) T$_H$ tries to get semaphore
(priority of T$_L$ is raised to T$_H$'s)

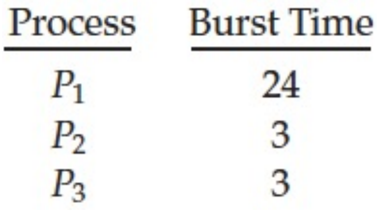
(8) T$_L$ releases the semaphore
(T$_H$ resumes)

(10) T$_H$ completes

Round Robin (RR) Scheduling

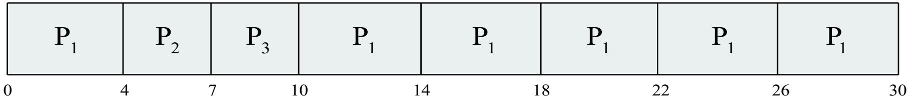
- Each process gets a small unit of CPU time (**time quantum *q***), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4



Process	Burst Time
P_1	24
P_2	3
P_3	3

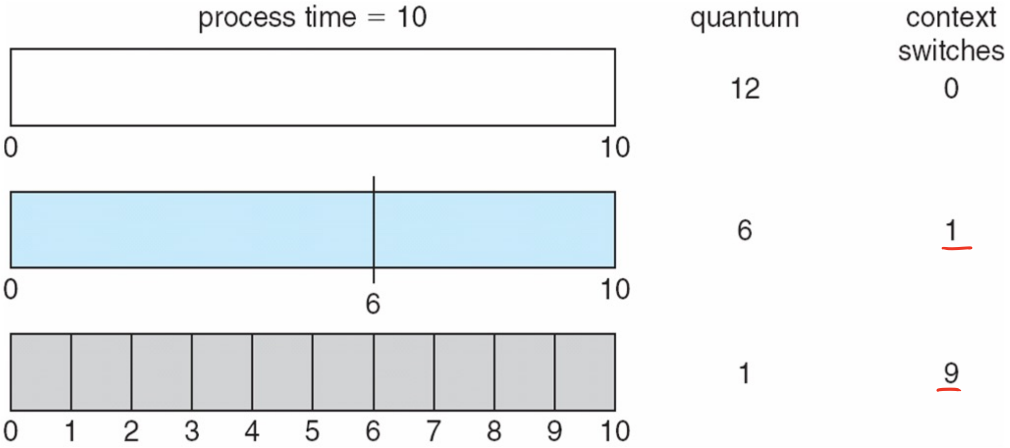
- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better ***response***

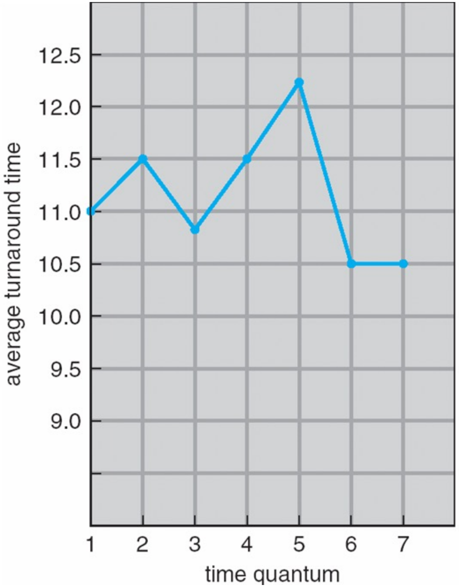
Time Quantum and Context Switch Time

- q should be large compared to context switch time; q usually 10ms to 100ms, context switch < 10 usec



quantum	context switches
12	0
6	1
1	9

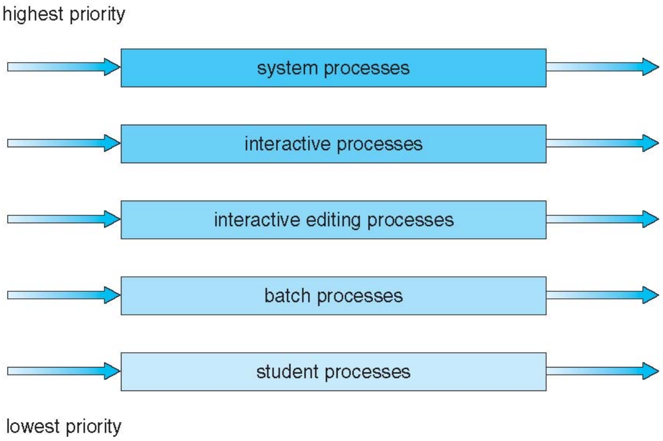
Turnaround Time Varies With The Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive) 대화형
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues: 선점형인것을생각. => back이 starvation의 가능성이 있음.
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR and 20% to background in FCFS



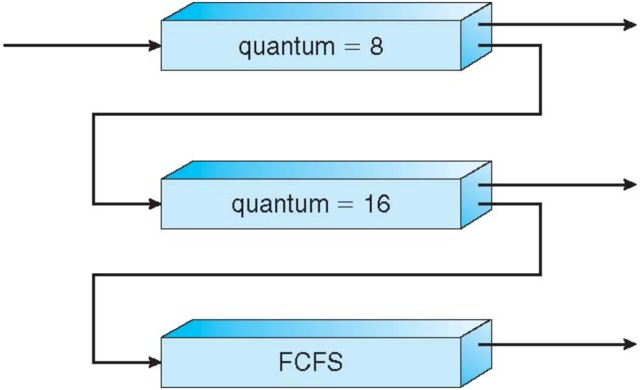
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds $x<8$
  - Q_1 – RR time quantum 16 milliseconds $8<x<16$
  - Q_2 – FCFS $16<x$

- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



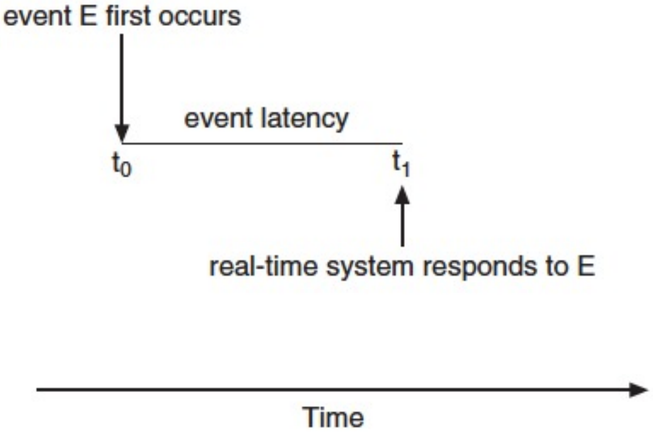
6.6 Real-Time CPU Scheduling

- Real-time system is event-driven
  - **Soft real-time systems** vs. **Hard real-time systems**

deadline이 엄격X

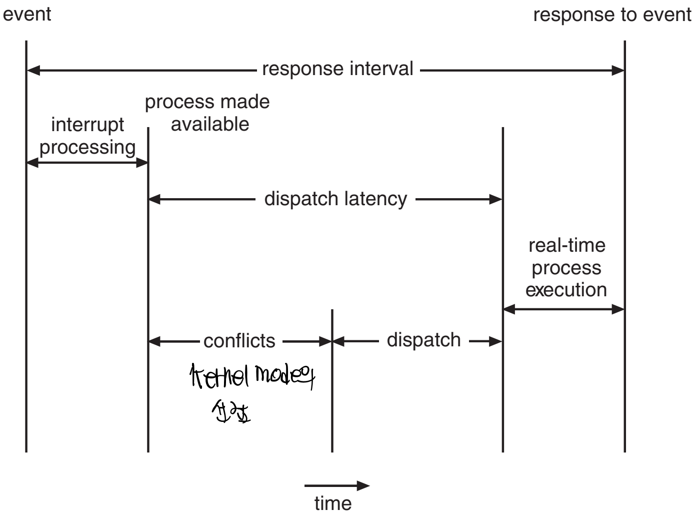
(deadline이 엄격)

- Event latency
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



Real-Time CPU Scheduling (Cont.)

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes

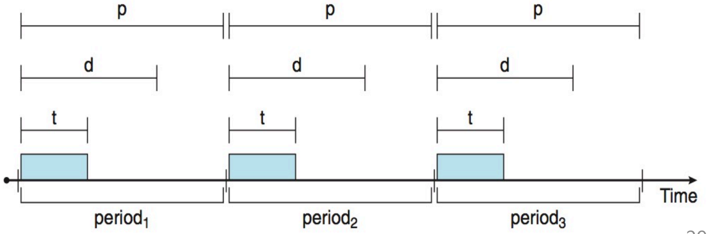


Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority-based scheduling → 즉각 대응해야하므로.
  - But only guarantee soft real-time

task = process

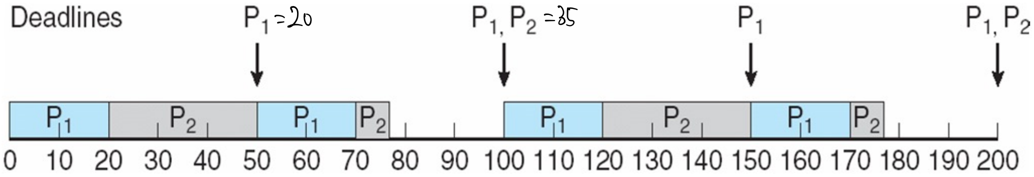
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals
  - Has processing time *t*, deadline *d*, period *p*
  - $\underline{0 \leq t \leq d \leq p}$
  - **Rate** of periodic task is 1/*p*



Rate Montonic Scheduling

static

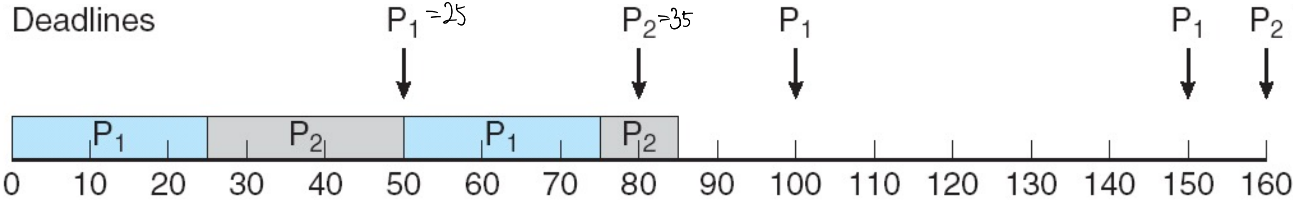
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority; Longer periods = lower prio rity
- P_1 is assigned a higher priority than P_2.



Missed Deadlines with Rate Monotonic Scheduling

정적에서 최적이다.

$P_1 = \frac{25}{50}$ $P_2 = \frac{35}{80}$.



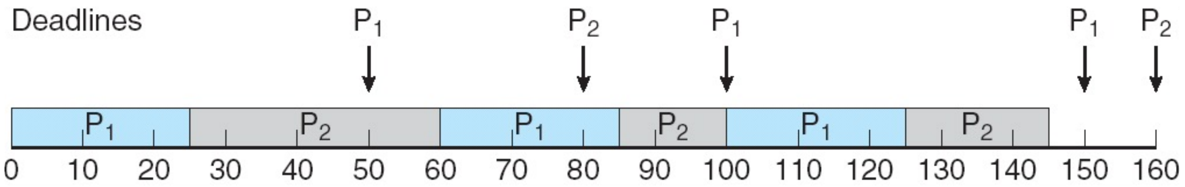
최악의경우의 cpu 이용률 = $N\cdot(2^{\frac{1}{N}}-1)$

Earliest Deadline First Scheduling (EDF)

Dynamic

- Priorities are assigned according to deadlines:

  the earlier the deadline, the higher the priority;
  the later the deadline, the lower the priority



Proportional Share Scheduling

- T shares are allocated among all processes in the system
- An application receives N shares where $N < T$
- This ensures each application will receive $\boldsymbol{N} / T$ of the total processor time

ex) 100 = T

50	15	20
A	B	C

6.7 Operating System Examples

- Linux scheduling
- Windows scheduling

Linux Scheduling Through Version 2.5

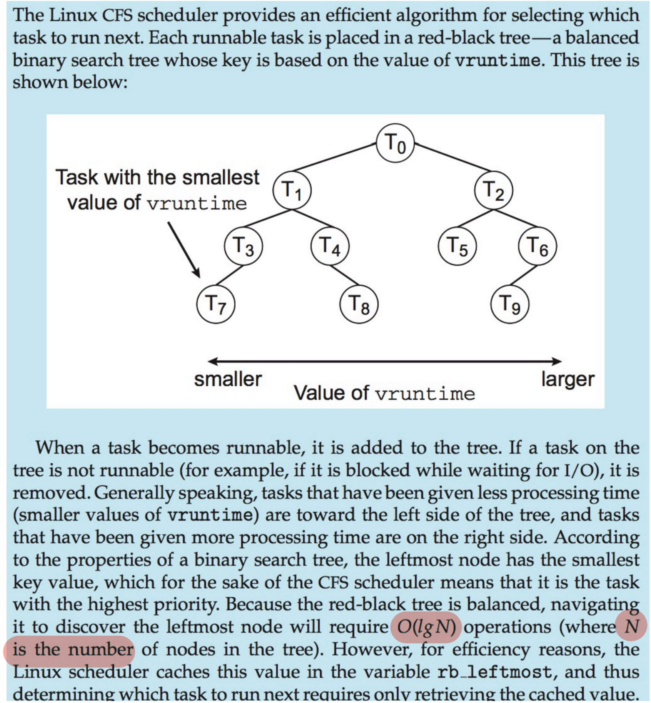
- Prior to kernel version 2.5, ran variation of standard UNIX scheduling algorithm
- Version 2.5 moved to constant order O(1) scheduling time
  - Preemptive, priority based
  - Two priority ranges: time-sharing and real-time
  - Real-time range from 0 to 99 and nice value from 100 to 140
  - Map into global priority with numerically lower values indicating higher priority
  - Higher priority gets larger q
  - Task run-able as long as time left in time slice (**active**)
  - If no time left (**expired**), not run-able until all other tasks use their slices
  - All run-able tasks tracked in per-CPU **runqueue** data structure
  - Worked well, but poor response times for interactive processes

Linux Scheduling in Version 2.6.23 +

- Completely Fair Scheduler (CFS)
- Scheduling classes
  - Each has specific priority
  - Scheduler picks highest priority task in highest scheduling class
  - 2 scheduling classes included, others can be added: default and real-time
- Rather than quantum based on fixed time allotments, based on proportion of CPU time 비율을 계산.
  - Quantum calculated based on nice value from -20 to +19
  - Lower value is higher priority
  - Calculates target latency – interval of time during which task should run at least once
  - Target latency can increase if say number of active tasks increases
- CFS scheduler maintains per task virtual run time in variable vruntime
  - Associated with decay factor based on priority of task – lower priority is higher decay rate
  - Normal default priority yields virtual run time = actual run time
- To decide next task to run, scheduler picks task with lowest virtual run time

CFS Performance

binary seach tree key: vruntime.



The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:

T_0
T_1
T_2
T_3
T_4
T_5
T_6
T_7
T_8
T_9
Task with the smallest value of vruntime
smaller
Value of vruntime
larger

When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(\lg N)$ operations (where N is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

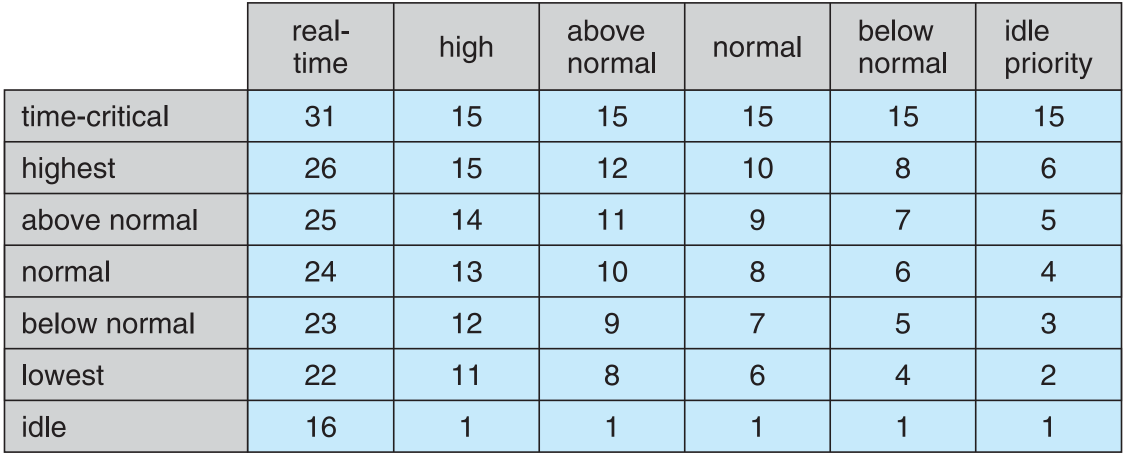
Windows Scheduling

- Windows uses priority-based preemptive scheduling
- Highest-priority thread runs next
- Dispatcher is scheduler
- Thread runs until (1) blocks, (2) uses time slice, (3) preempted by higher-priority thread
- Real-time threads can preempt non-real-time
- 32-level priority scheme
- Variable class is 1-15, real-time class is 16-31 0은 memory 관련.
- Priority 0 is memory-management thread
- Queue for each priority
- If no run-able thread, runs idle thread

Windows Priority Classes

- Win32 API identifies several priority classes to which a process can belong
  6개 ↓
  - REALTIME_PRIORITY_CLASS, HIGH_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS,NORMAL_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, IDLE_PRIORITY_CLASS
  - All are variable except REALTIME
- A thread within a given priority class has a relative priority 6개의 class 안에 7개
  - TIME_CRITICAL, HIGHEST, ABOVE_NORMAL, NORMAL, BELOW_NORMAL, LOWEST, IDLE ⇒ 7개
- Priority class and relative priority combine to give numeric priority
- Base priority is NORMAL within the class
- If quantum expires, priority lowered, but never below base

Windows Priorities



	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1